A Component-Based Middleware for a Reliable Distributed and Reconfigurable Spacecraft Onboard Computer

Ting Peng, Kilian Höflinger, Benjamin Weps, Olaf Maibaum
Simulation and Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
{Ting.Peng, Kilian.Hoeflinger, Benjamin.Weps, Olaf.Maibaum}@dlr.de

Kurt Schwenk
German Space Operations Center (GSOC)
German Aerospace Center (DLR)
Oberpfaffenhofen, Germany
Kurt.Schwenk@dlr.de

Daniel Lüdtke, Andreas Gerndt
Simulation and Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
{Daniel.Luedtke, Andreas.Gerndt}@dlr.de

***Abstract*—Emerging applications for space missions require increasing processing performance from the onboard computers. DLR's project "Onboard Computer - Next Generation" (OBC-NG) develops a distributed, reconfigurable computer architecture to provide increased performance while maintaining the high reliability of classical spacecraft computer architectures. Growing system complexity requires an advanced onboard middleware, handling distributed (real-time) applications and error mitigation by reconfiguration. The OBC-NG middleware follows the Component-Based Software Engineering (CBSE) approach. Using composite components, applications and management tasks can easily be distributed and relocated on the processing nodes of the network. Additionally, reuse of components for future missions is facilitated. This paper presents the flexible middleware architecture, the composite component framework, the middleware services and the model-driven Application Programming Interface (API) design of OBC-NG. Tests are conducted to validate the middleware concept and to investigate the reconfiguration efficiency as well as the reliability of the system. A relevant use case shows the advantages of CBSE for the development of distributed reconfigurable onboard software.**

Keywords—middleware; reliability; distributed system; component-based software engineering; reconfiguration; spacecraft

I. INTRODUCTION

Onboard systems for space missions are growing in complexity. In order to decrease the complexity of application development and to support the future reutilization of existing programs, the onboard system software needs to facilitate reusable and modular development. Another fundamental requirement for onboard system software is its real-time capability, due to the fact that some applications are time-critical and require meeting specific execution deadlines.

After launching into space, it becomes difficult or even impossible to repair the spacecraft when parts of it fail. Therefore, the reliability of the spacecraft is vital for the entire lifetime [1]. Spare components, invoked by redundancy configurations are usually used to increase reliability.

The onboard computers for the space environment need to be radiation robust, since they are exposed to energetic particles which may lead to Single Event Upsets (SEU) [2], etc. Usually, radiation-hardened computers are mainly used for space missions, high-altitude aircrafts, etc. These radiation-hardened products are far more expensive and also less powerful than hardware for the industry market. The emerging utilization of Commercial Off-The-Shelf (COTS) hardware offers cost reduction for space mission development. On the one hand, COTS components provide higher processing performance, compared to radiation-hardened computers. On the other hand, they face the problem of damage and malfunctions due to space radiation.

The project "Onboard Computer - Next Generation" (OBC-NG) takes advantage of multi-core COTS processors, which offer high computing performance compared to standard spacecraft processors. OBC-NG's architecture is based on a distributed networked reconfigurable system. It uses a new redundancy approach to gain high reliability [3] and supports a multi-core version of the Real-time Onboard Dependable Operating System (RODOS) to satisfy hard real-time requirements for time-critical applications [4]. Linux is an additional selectable operating system to enable the use of third-party libraries for complex applications, if needed. OBC-NG aims to offer high performance, reliability and redundancy for applications in the space environment.

Complex onboard software in the space domain usually consists of an operating system (OS), a hardware abstraction layer, a middleware and applications for the attitude and orbit control subsystem, propulsion subsystem, power subsystem, communication subsystem, scientific payloads subsystem, etc. The middleware has the role of a data handling service, a task management service, a monitoring service, a reconfiguration service, peripheral control and communication service.

The OBC-NG middleware (see Fig. 1) is a distributed networked framework, which acts as the provider for task-oriented Application Programming Interfaces (APIs), and modular distributed components for model-driven software development. To support transitions among different phases of a space mission and recovery from failures or errors, the task management service, the monitoring service, and the reconfiguration service are present in the OBC-NG middleware.

The final publication appeared: T. Peng et al., "A Component-Based Middleware for a Reliable Distributed and Reconfigurable Spacecraft Onboard Computer," 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS), Budapest, Hungary, 2016, pp. 337-342. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7794363&isnumber=7794313

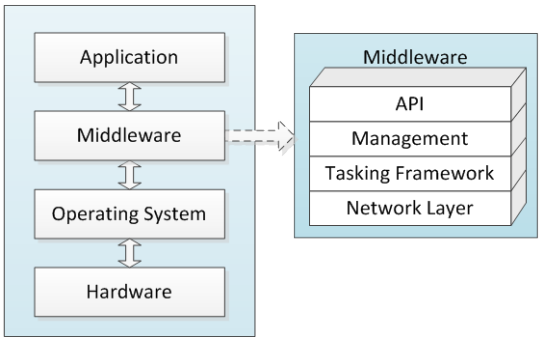


Fig. 1. OBC-NG system and middleware architecture

Classic redundancy concepts usually have a one-to-one mapping among components and their redundant counterparts. In contrast, OBC-NG does not assign specific nodes as redundant counterparts. Tasks can be moved to all compatible and available nodes. The currently used nodes consist of a CPU and an FPGA. It is also planned to support task morphing from software to hardware and vice versa [5].

In the context of OBC-NG, reconfiguration means that the deployment of tasks and services can be restructured according to different predetermined configurations. The reconfiguration targets the software and the hardware [3] as well as the network routing. OBC-NG considers two main reconfiguration types: planned reconfiguration and reconfiguration due to a failure [3]:

- Planned reconfiguration: different mission phases, such as the descent and landing phase of a spacecraft and the following scientific ground operation phase on a celestial body, require different configurations. In each phase, different tasks with different sensors, actuators, and scientific instruments are conducted [3].
- Reconfiguration due to a failure: when a failure occurs on a node, a reconfiguration is triggered to isolate the faulty node. Then the tasks on this node need to be migrated onto other properly functioning nodes.

To facilitate the development of this kind of distributed framework, the Component-Based Software Engineering (CBSE) is selected. The CBSE is a software development approach that is based on software reuse. The CBSE process can be classified as component development and system development with software components [6]. In component design, concerns (which are interests or focuses of information that can affect the development of the software) are separated and functionalities are decoupled to fit into the component model. The system development mainly involves development of components, system requirements studies, and the selection of available components to meet the system requirements. The system requirements include requirements of standardization, independency, composability, deploy ability, and documentation. The selected components may need to be adapted to fit the requirements. After the selection and adaptation of components are fixed, components are assembled and tested on the system platform [7]. Ishita Verma [6] proposes the W model for CBSE development. In the W model, domain-engineering techniques are considered for the component development.

During onboard software development, model-based techniques together with domain expert knowledge can be integrated into component design and implementation. Diagrams or Domain-Specific Languages (DSLs), besides normal programming languages, are used for modeling [8], code generation, software verification, and design validation [9]. Meanwhile, the middleware that adopts CBSE can facilitate planning, mapping, and optimization of tasks and configurations in a model-based way.

This paper demonstrates that the concept of CBSE is especially helpful for the development of distributed and reconfigurable onboard software. It also shows how to implement the composite component framework for the middleware and applications. By using of the composite component framework, the modules of OBC-NG middleware and applications are easy to implement, adjust, and reuse.

The reminder of the paper is structured as follows. Section II presents related work. Section III describes the architecture of OBC-NG's middleware. Section IV details the composite component framework, which is adopted by the OBC-NG middleware, including the metamodel, modular distribution, and model-based development, test and verification. Section VI evaluates the middleware and gives a usage scenario. Section VII concludes the paper and presents an outlook for future work.

II. Related Work

In several embedded system domains, including not only space applications but also robot technology, similar middleware architectures are utilized.

In aerospace domain, the demands of task management, health monitoring, fault tolerance and parallel processing lead to the evolution of middleware. These demands are separated from and atop operating systems. The Adaptive Dependable Distributed Aerospace Middleware (ADDAM) has been developed for the flight system management over multiple computers [10]. ADDAM is designed, being portable and reusable. It adopts event-based data receiving. Dependable Multiprocessor (DM) leveraged high-performance COTS processors to increase the performance of the onboard computers [11]. Its middleware, called Dependable Multiprocessor Middleware (DMM) handles tasks, failure detection and recovery. DMM also provides a platform-independent API. DMM's APIs support heartbeating and checkpointing for applications.

In robotics, middlewares usually abstract lower layer details and offer easy-to-use interfaces or a modular framework for high-level applications and implementations. A distributed component middleware called RT-Middleware (Robotics Technology Middleware) has been developed to improve the reusability by offering a modular software structure and to ease the complexity of integration [12]. Real-time ability is a requisite for robots to react to the actual environment within certain deadlines. RT-Middleware introduces a new composite component called RT-Component for independent low-level and real-time composition [13]. Player 2.0 is a robot programming framework which simplifies the driver API and hides most parts of the communication, thus eases maintenance [14]. Christian Schlegel et al. [15] define explicitly stated properties within components for model-driven designs for robotic software systems. As can be seen from [16], many robotic middlewares aim at not only improving the reusability

and flexibility but also at decoupling robotic software design and implementation through modular structure platforms and component-based development approaches, such as Orocos, Orca, OpenRTMaist (that is the implementation of RT-Middleware), MARIE, SmartSoft, etc.

In the distributed system domain, the distributed computing middleware CORBA has been widely used to address the challenges of heterogeneity, network-centric operation and dynamic operating conditions [17], [18], [19], [20]. Its interfaces are used at mission control centers for spacecraft operations [21], [22]. However, CORBA's target market resides in the commercial area and it is not suitable for real-time, fault-tolerant and reconfigurable onboard systems in which low-memory-footprint programs are required [23].

In addition, the concept and methodology of CBSE has been applied in both robotics and the development of onboard systems for spacecraft. In the space domain, CBSE has been integrated into the development of onboard software e.g. [24], [25], [26]. The project ASSERT [25] is a good example of the combination of CBSE methodology and onboard software. Marco Panunzio et al. [25] summarized the requirement and feasibility analysis from European Space Agency (ESA) projects and how to take advantage of CBSE for onboard software reuse. Marek Prochazka et al. [27] followed the CBSE methodology to establish a component-oriented framework for onboard software with dynamic reconfiguration of applications. They aim at improving reusability of software for different space missions and easing complexity of development and integration. SOFA HI [27] offers a metamodel for application component design. The component-based concept is also used in unmanned aircraft systems (UAS) to deal with increasing complexity of subsystems, distributed communication and management services [28]. Marco Panunzio et al. [26], [29] designed a domain-specific metamodel called Space Component Model (SCM) for component models for ESA research and development projects.

III. Overview of the Middleware Architecture

The OBC-NG middleware is designed as a layered architecture. It consists of a network layer, the Tasking framework and a management layer to offer communication services, application task services and management as well as monitoring services (see Fig. 1). The OBC-NG middleware uses message-triggered and event-triggered mechanisms for task execution and management. The middleware supports the real-time operating system RODOS and Linux for the consideration of two aspects, i.e., some applications require the hard real-time abilities of RODOS [3] and some applications rely on third-party Linux libraries. The features including management, monitoring, reconfiguration and model-based development are implemented in the OBC-NG middleware.

A. Network Layer

The network layer is responsible for the communication among nodes in the distributed onboard system. It is visualized in Fig. 2, incorporating network protocol, network connector, underlying protocol, event handler and timer service [30]. The network protocol is the core component of the network layer, which transmits and receives messages of different transmission types in the network. The network connector is an abstraction to the transport layer to transmit and receive messages through the underlying transport protocol. The event handler handles the received data and is triggered by the network protocol. The timer service is used to invoke the timer functionality of the hardware. Currently, the underlying protocol supports Ethernet with UDP/IP. The next step is to integrate the SpaceWire Protocol, since it is widely used in the space domain. SpaceWire is a network connection that is low-latency, full-duplex and is based on point-to-point serial links [31]. The OBC-NG network layer supports the transmission of unreliable data, reliable data, request data, response data, reconfiguration commands, message acknowledgements, heartbeats, error notifications and large-size messages. Moreover, subscription and broadcast mechanisms can be realized by using the network layer. Furthermore, the network layer is also designed to support monitoring, error detection and reconfiguration on higher layers.

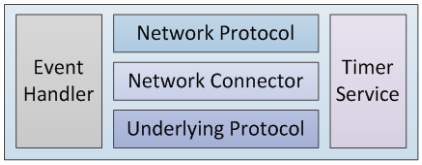


Fig. 2. Network layer structure (evolved from [30])

B. Tasking Framework

The Tasking framework offers application task services [32]. The Tasking concept has its origin in RODOS and serves as a hypervisor. It provides communication and scheduling capabilities for task-based applications. To use the Tasking framework, application developers need to divide their algorithms into smaller tasks. These tasks can then be distributed on several nodes or cores on a multi-core CPU. Within the Tasking framework, a task has three actions, i.e., consuming information, performing computations, and producing information, which can be used as input message by other tasks. The task computation is triggered either by an event or by the fact that all required input data have been collected. The results of computations are information which can be used by other subsystems or modules [32]. The Tasking framework can be used for distributed and shared-memory system architectures. The communication media are messages and events. Workload partitioning as well as task mapping needs to be realized explicitly. The Tasking framework offers thread management and synchronization and it has been used for several space missions at DLR.

C. Management Layer

The management layer offers task distribution, monitoring and reconfiguration services for nodes in coarse granularity and tasks in fine granularity. The management layer has four tasks: monitor, reconfiguration manager, reconfiguration service and checkpointing service. For the project OBC-NG, nodes can be either of the type Processing Node (PN) or Interface Node (IN). Computation and management tasks run on the Processing Node. Thus the role of PN can be Master (M), Observer (O) or Worker (W). As the onboard system includes various peripheries such as sensors, actuators, instruments, and mass storage, the Interface Node is the connection part between the network and peripheries. Therefore the role of IN can be Storage (S) or Interface (I) [3]. The IN is also responsible for the management of data subscription lists, i.e., the periphery

sensor will send acquired data to the tasks, which are registered in the subscription list of this periphery sensor.

At the end of each iterative round of application task execution, the checkpointing service sends checkpoint values, i.e. snapshots of states of tasks to the Storage node.

Both Master and Observer use the monitor service to send heartbeats to other nodes to detect whether one node is running nominally or not. The monitor can also specify threshold values and determine whether an application task still behaves nominally or not by comparing the control value responded from the node running this application task.

When a failure is detected by the Master or informed by Workers or Observers to the Master, the Master will use the reconfiguration manager to search a decision graph for a suitable configuration according to the failure information (see Fig. 3). The Master broadcasts the reconfiguration command to other nodes to trigger the reconfiguration on node level. When no practicable configuration is found, a safe mode is triggered by the Master which is handled only by the Master itself. In safe mode, nonessential components are switched off and only safety critical components are running. The reconfiguration manager is also responsible for triggering the new phase reconfiguration when a spacecraft enters a new mission phase.

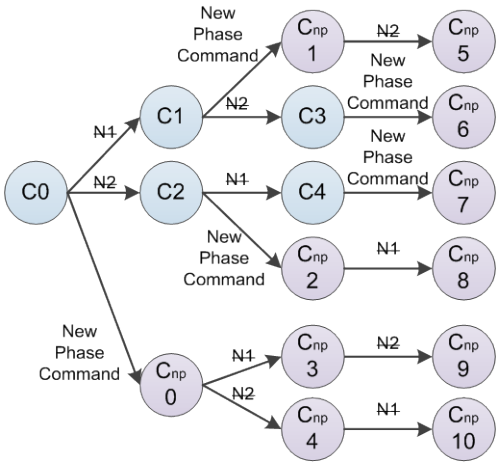


Fig. 3. Simple example of a decision graph to mitigate node failure: Cx denotes a configuration, $C_{np}x$ denotes a configuation in the new mission phase and ~~Nx~~ denotes a failing node (evolved from [3])

All nodes, including the node with the Master, use the reconfiguration service to perform the initialization of management and application tasks on corresponding nodes. During reconfiguration, if an application task uses the checkpointing service, the nodes with this kind of application tasks will request the checkpoint values which are periodically saved to a Storage node.

IV. Composite Component Framework for Application Software and Middleware

To facilitate the distribution and combination of application tasks on multiple nodes, the composite component framework, which derives from CBSE, is developed for the OBC-NG middleware and the application layer.

A. Metamodel for the OBC-NG Middleware

To form the composite component framework for application software and the OBC-NG middleware, a metamodel is presented to describe and normalize the elements of the framework that includes component models and component interfaces, the structural relationship among elements, and the component models' hierarchy of the composite component.

1) Component Model: Components can be treated as a service provider for calculations or operations. In OBC-NG, the core functionality of a component is realized by a task. The data types used by a component's task, input task messages, and output task messages should contain the message destination's node logic address together with the port number and transmission type that is limited to unreliable data, reliable data, pull request, and pull response for application tasks. A basic component consists of a task (either an application task or a management task), input channel, and output channel.

2) Component Interface: The communication among components is implemented via the component interface. Component interfaces can be divided into two categories, i.e., required interfaces and optional interfaces. Only when all services connected to required interfaces are available or collected, the component itself can then fulfill its own service.

The input channel is the component interface that requires messages as inputs or is triggered by events for task execution. Each input comprises a task reader, a typed task message and a task input from the Tasking framework. The task reader is triggered by the event handler of the network layer when the corresponding port receives data.

The output channel is the component interface, which provides messages and events as outputs for other component's tasks. Each output comprises a typed task message and a task writer from the Tasking framework. When the task writer is notified by the output task message, it will send messages out through the interfaces offered by the network layer.

Due to the consideration that components can run on different PNs and the same component can run on different PNs under different configurations, it is necessary to unify the component interfaces.

3) Composite Component Framework: As for a composite component, several basic components can be composed into a composite component if necessary (see Fig. 4).

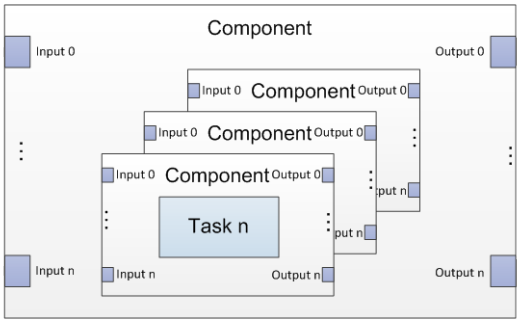


Fig. 4. Composite component

A message is delivered to its destination according to the port number and the node logic address (see Fig. 5 for an example). It is a point-to-point message passing on both node level and component level. When a message reaches its

destination node, only the component with the corresponding port number will be triggered for execution, i.e. the message will enter the component through the input channel.

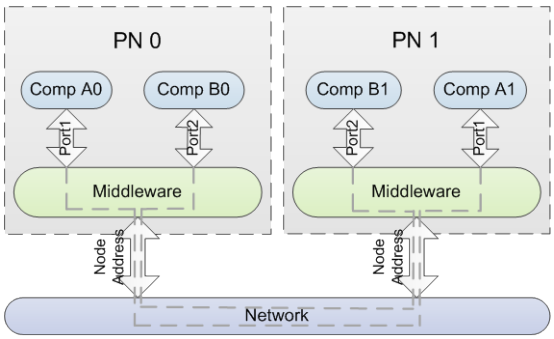
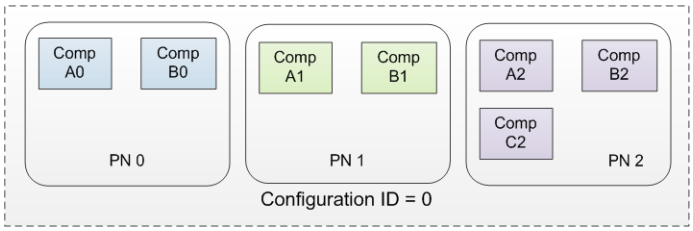


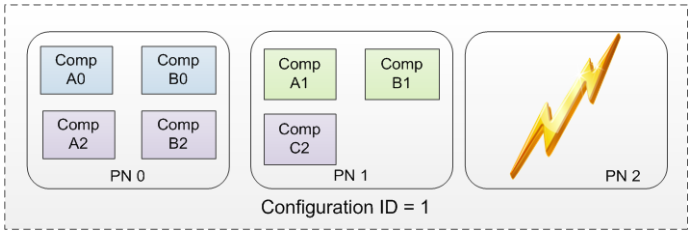
Fig. 5. Message transmission in the network

B. Modular Distribution of Components

Both application tasks and management tasks are realized as components. Thus, the configuration settings are used to specify which task should run on which node for each configuration ID. Tasks are stored locally on each Processing Node so that only the new configuration ID is needed to broadcast to each node during reconfiguration. Fig. 6 shows an example of the component distribution on different PNs with different configuration IDs. When the failure occurs on PN 2, components running initially on PN 2 need to deploy on other properly working nodes. The middleware support to the modular distribution of applications is given below and the component reusability is considered for further development.



(a) Component distribution of the initial configuration



(b) Component distribution of the configuration after PN 2 fails

Fig. 6. Distribution of components

1) Middleware Support: In order to increase the reliability of distributed nodes and to make the system fault-tolerant, the middleware offers several services to support components running on a PN (see Fig. 7). These support services are described as management layer in Section III.

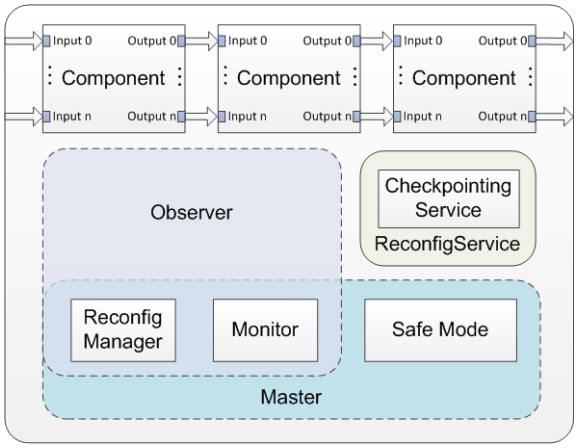


Fig. 7. Middleware support on a PN

2) Reusability Considerations: In order to produce reusable software, the API design and the utilization of the CBSE concept are beneficial not only for component reuse but also for the model-driven application software development with code generation techniques. Model-based code generation is becoming increasingly popular and important in the space onboard software development area. Meanwhile, the methods and elements in components need to enhance generality. The models adopting the CBSE are easier to be integrated and configured for different space missions.

CBSE covers not only the software architecture, modular software design, configuration, and deployment. It also considers software verification [35]. From an efficiency point of view, tests and verifications of components can be decoupled and reused with little or even without modifications.

C. Model-Based Development, Test and Verification

Model-based approaches, which use diagrams or DSLs for modeling and model-based systems engineering, are also beneficial for the composite component development for the middleware and application programs. Model-based approaches besides CBSE can largely reduce human faults and therefore improve the safety of onboard application software.

With the help of CBSE, onboard task and mission planning can generate automated plans and schedules, then schedule and execute these plans on component level.

When developing and reusing components for different missions, not only the component itself should be well tested but also the component for a mission should be tested in the corresponding environment [36]. In other words, components should offer implementation and interfaces for test and verification for a specific mission. Recently, the model-driven architecture was applied for component testing [37].

As Marek Prochazka et al. [27] mention, when a system is divided into several composite components and basic components properly, the complexity of verification can be largely reduced. It is the application programmer's responsibility to well divide their applications into components.

V. Middleware Implementation

After establishing the OBC-NG middleware architecture and the composite component framework for application software and the middleware, application programmers should utilize the

API defined in OBC-NG middleware to build their application on top of the middleware layer. The concept and methods of using OBC-NG middleware for applications are explained in the following subsections. In the future, a graphical modeling tool with code generator will be provided to set up the system and applications.

A. Application Programming Interface Design

In order to facilitate and match the model-driven development, an application task should define the following interfaces as shown in Fig. 8. First the internal states are set up when the Tasking framework is initialized. Tasks are activated on nodes. The OBC-NG middleware offers interfaces to send and receive messages respectively. When the activated task is triggered by events or messages, it starts the execution to perform its specific task defined by the user and calculate the outputs resulting from inputs and parameters of the current configuration. The states and the control values are updated and outputs are sent out to their destinations. After the execution, a snapshot is taken and sent to the Storage node for the checkpointing service. The component enters idle state again.

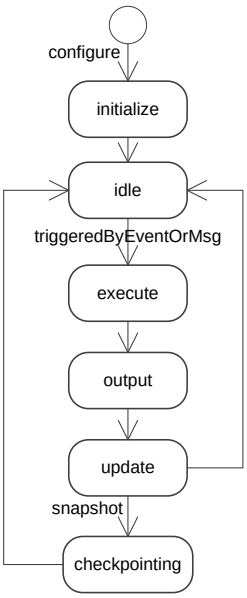


Fig. 8. State diagram of the component

Inputs should specify the port number on which the message arrives. Both inputs and outputs contain the data which are defined as input and output data for each computation step respectively. The states will be used as snapshot content for the checkpointing service and reconfiguration. The control value is set by users to check the plausibility of some application tasks' internal states and output values. Different parameters are specified since different mission phases or different configurations may require different parameters for the task calculation.

B. Middleware Configuration File

The middleware configuration file defines the placement of a task on the distributed system. For each configuration ID, it defines the placement of application tasks and management tasks, node health state, application task ID, storage location, and port number as well as the decision graph for this mission phase. Different configuration files can be defined for different mission phases. The configuration ID of new mission phase will be broadcasted to all nodes in the network with enough lead time for the preparation of the new mission phase. The OBC-NG middleware requires this information for management and application tasks deployment, activation, and deactivation. During the reconfiguration, subscriber lists of the IN need to be updated according to the information in the configuration file. In this way, during either initial configuration or reconfiguration, only the configuration ID needs to be broadcasted to each node.

VI. Evaluation and Performance Analysis

In this part, the OBC-NG middleware, designed and implemented with the component-based approach, is evaluated. The focus of the evaluation is the reconfiguration efficiency and the implementation overhead. A usage scenario is given to demonstrate the practicability of the OBC-NG middleware concept.

Both non-real-time reconfiguration and real-time reconfiguration are based on the CBSE support.

A. Non-Real-Time Reconfiguration

Two different situations are tested, i.e. non-real-time new phase reconfiguration and non-real-time node failure reconfiguration.

For non-real-time new phase and node failure reconfigurations, the validation of the framework was carried out on a prototype with three Processing Nodes (Xilinx Zynq Z-7020) and one Storage node (Xilinx Zynq Z-7010). All PNs run PetaLinux (Kernel version: 3.17). A Master, a higher-priority Observer, a lower-priority Observer and two application tasks are distributed on these three PNs. All PNs and the Storage are linked via a router. Both non-real-time new phase and node failure reconfigurations are repeated for 200 times and 150 times respectively. The reconfiguration costs are evaluated for heartbeat periods of 100ms, 500ms, 1000ms, 2000ms, 3000ms and 4000ms. Especially for the node failure reconfiguration, different seeds are used to generate random failure on different PNs. For non-real-time reconfiguration, the requirement for time bound of the reconfiguration costs is set as 5s.

The results in Fig. 9 show that the new phase reconfiguration time is not affected by the heartbeat period. The new phase reconfiguration costs for different heartbeat periods are between 611ms and 935ms with the average cost of 807ms. The standard deviation of 57ms means that the measured reconfiguration costs are stable.

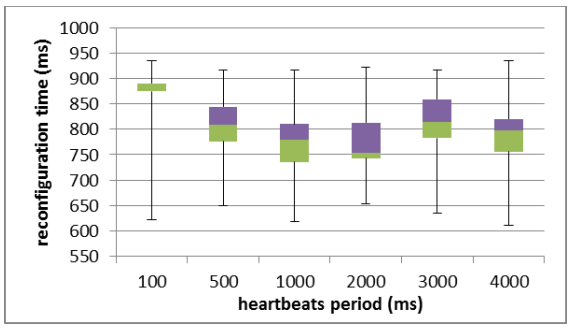


Fig. 9. New phase reconfiguration costs

The results in Fig. 10 show that the reconfiguration time increases as the heartbeat period increases. But all reconfiguration times satisfy the 5s time bound of the requirement.

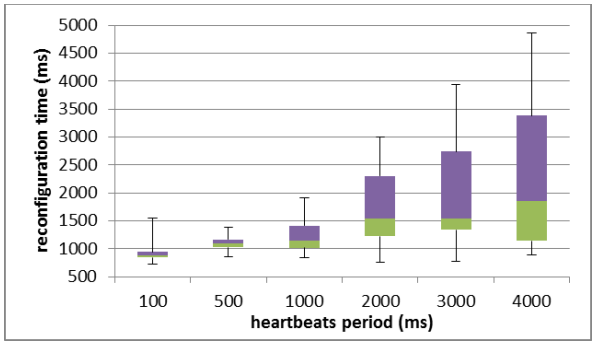


Fig. 10. Node failure reconfiguration costs

B. *Real-Time Task Reconfiguration*

For real-time task reconfiguration, the costs of changing the subscription list of sampling data for real-time tasks are measured in the following situation: Sampling task A and task B publish data at the frequency of 20ms. Task C subscribes data from task A. At some point, task A fails so that the task C subscribes data from task B instead. Task A and task B run on two PNs of PetaLinux respectively and task C runs on the RODOS real-time OS. For the real-time tasks, the switch of the subscription list after task A fails should be within 100ms. And the average switch time for 150 runs is 66ms (see Fig. 11).

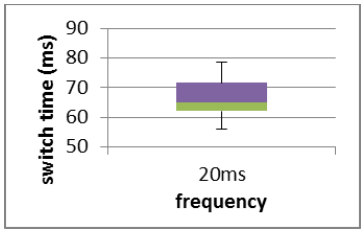


Fig. 11. Real-time task switch costs

C. *Framework Evaluation*

Overhead of the framework implementation comes from the following aspects. There is data encapsulation overhead, which contains necessary information for data delivery to ports on distributed nodes such as the message type and destination address including the node logic address and the port number. The overhead of data encapsulation for unreliable data transmission, reliable data transmission, data request, data response, reconfiguration request, acknowledge, heartbeat and error notification is 17 Byte. The overhead of data encapsulation for large message transfer (>54 KByte and <1 MByte) and large file transfer (>1 MByte) are 24 Byte per 54 KByte and 420 Byte per 1 MByte respectively.

In terms of memory footprint, the overhead of the composite component framework, the network layer, the Tasking framework, and the management layer can also affect the size of the memory footprint. Legacy codes such as the codes of Attitude and Orbit Control System (AOCS) need to be converted to fit into the composite component model. In order to measure the trend of memory consumption, we increased the number of instantiated components. For all instantiated components, all tasks had empty execution function. The results of hard-disk memory usage are presented in Fig. 12.

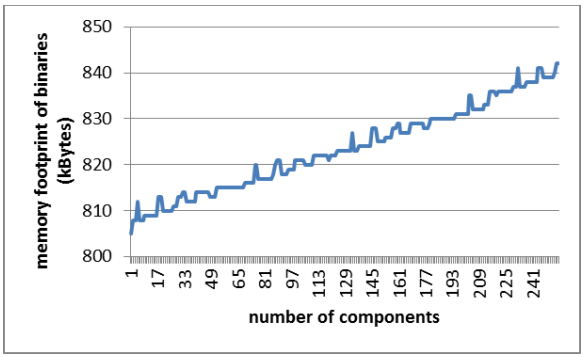


Fig. 12. Memory footprint of binaries

We also measured the configuration cost when the number of components scales to investigate the influence of the amount of components. The reconfiguration time is measured from the time the initial configuration command is triggered to the time all components finish the initial configuration. As can be seen from Fig. 13, the configuration time is slightly increasing with increasing number of components. The current version of OBC-NG middleware only supports up to 256 components on a single node.

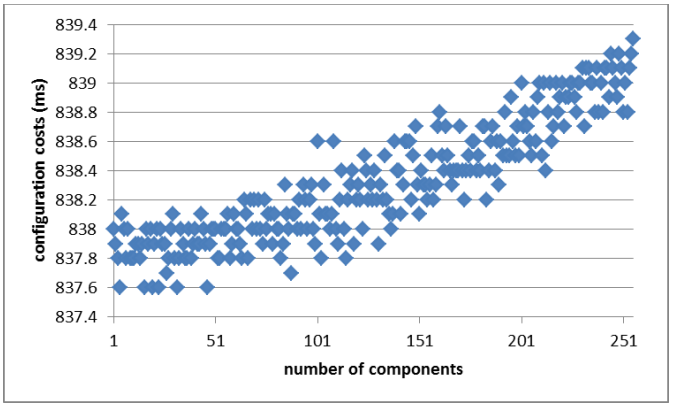


Fig. 13. Configuration costs for different numbers of components

As all tasks are implemented as components, the communication and data exchange is based on message transmission. The average transmission costs for different numbers of components are all within 6ms. This means that the overhead of message transmission for the component-based middleware is very small.

D. *Earth Observation*

The OBC-NG middleware is suitable for many space application scenarios that require high onboard computational and processing performance.

For an earth observation application, it is not realistic to downlink all raw images and data to the ground station. It is better to first calculate the coverage of clouds on the image. If there are too many clouds on the image, i.e., it contributes little scientific value for earth observation, it will not be transferred to the ground. This processing of high-resolution images also

requires high computational performance of the onboard computer.

The following earth observation demonstration shows a simplified version of the standard ACCA-cloud detection algorithm [38], which is implemented using the OBC-NG middleware framework. Four PNs are used for this demonstration with additional monitoring and failure reconfiguration functionalities. The sensor task is used to act as taking images by the cameras and sending the images to ACCA tasks. To exploit the parallel processing architecture of the OBC-NG system, two other nodes are used for the image processing, each running a complete identical instance of the cloud detection algorithm (AccaTask0, AccaTask1). The images are processed in an alternate order. Finally, the processed images are transferred to a desktop to display the results on the monitor (see Fig. 14).

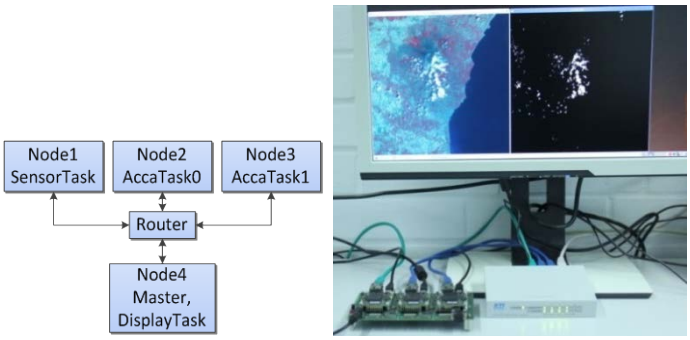


Fig. 14. Earth observation applications

If one of the processing nodes fails, the failure is detected by the Middleware and the system is reconfigured using the healthy node for processing. The failure mitigation is completely managed by the Middleware, without involving the applications (AccaTasks).

The four channel 8-Bit 2048*2048 pixels Landsat ETM+ 7 images used for the demonstration have a size of around 16 MByte each. Due to the dependency of the OpenCV and Boost libraries, the PetaLinux is chosen as the operating system on the OBC-NG board. With the support of the OBC-NG middleware API, the layout and the relationship of the earth observation application are straightforward and easy for the reconfiguration. Currently, the space application programmers need to care about settings for component interfaces such as input and output channels. An improved method to eliminate this is using tools for modeling and code generation of component interfaces.

The binary on each node of the OBC-NG board has a size of 778 KByte excluding the application. When taking the application into account, the binary size increases to 2.8 MByte. Compared to the applications, the overhead of memory footprint for the middleware is rather small (accounting for 28%).

The performance and redundancy are tested in this scenario demonstration. The time for the transmission of one test image (16 MByte), processing this image, and displaying the cloud mask on the screen is 5.136s on average. After the node with AccaTask1 failed, this image period increased to 9.608s on average (see Fig. 15). With the component-based approach, we can extend and upgrade the earth observation system easily by integrated new components.

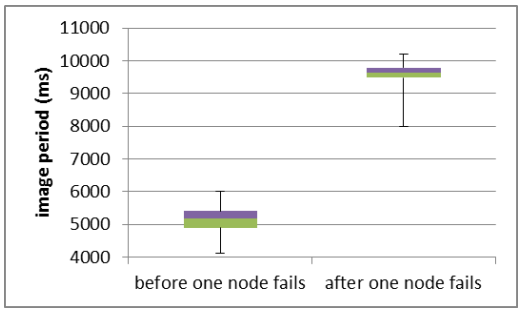


Fig. 15. Image periods before and after one node fails

VII. Conclusions

In this paper, we proposed a middleware architecture to support reusable and model-driven application software development in distributed onboard computers and to offer high reliability and high performance. This paper gives an overview of the middleware architecture which consists of a network layer, the Tasking framework, and the management layer. The management layer of the middleware offers application program management, monitoring, checkpointing, and reconfiguration services.

The Component-Based Software Engineering (CBSE)'s concept and methodology were adopted for the development of composite component framework for application software and the management layer of the middleware. In order to utilize CBSE, the component interface and the component model were defined and implemented. To use the composite component framework, the middleware support was specified. Applications were distributed modularly in the network according to the configuration. The reusability, test and verification constraints, and overhead of composite components were discussed in this paper as well. Finally, the earth observation application was carried out to verify our concept and mechanism of the middleware. During the implementation of the OBC-NG middleware, CBSE simplified the structure of the management layer. Modular services including management, monitoring, checkpointing, and reconfiguration in the management layer were easy to realize, adjust, and reuse. The models which adopted the CBSE were easier to be integrated and configured. The OBC-NG middleware showed the advantages of utilizing CBSE for the reconfigurable onboard software for spacecraft.

In the future, we will refine the monitoring mechanism, replace UDP with SpaceWire and enhance processing performance. A GUI for application design is also planned to be implemented. A tool for model-based systems engineering which is called Virtual Satellite [39] will be considered for the component-based applications development for spacecraft.

Acknowledgments

We would like to thank all the members of the OBC-NG project team at DLR in which the development of the middleware is carried out.

References

[1] D. Kim, S. Lee, J.-H. Jung, T. Kim, S. Lee and J. Park, "Reliability and availability analysis for an on board computer in a satellite system using standby redundancy and rejuvenation," *Journal of Mechanical Science and Technology*, vol. 26, no. 7, pp. 2059-2063, 2012.

[2] "Space radiation effects," [Online]. Available: http://www.xilinx.com/esp/aerospace-defense/space/radiation-effects.htm.

[3] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, T. Peng, B. Weps, G. Fey and A. Gerndt, "OBC-NG: towards a reconfigurable on-board computing architecture for spacecraft," in *Proceedings of IEEE Aerospace Conference*, Big Sky, Montana, 2014.

[4] S. Montenegro and F. Dannemann, "RODOS-real time kernel design for dependability," in *Proceedings of DASIA 2009 Data Systems in Aerospace*, Noordwijk, Netherlands, 2009.

[5] C. Haubelt, D. Koch, F. Reimann, T. Streichert and J. Teich, "ReCoNets—design methodology for embedded systems consisting of small networks of reconfigurable nodes and connections," in *Dynamically Reconfigurable Systems*, M. Platzner, J. Teich and N. Wehn, Eds., Springer Netherlands, 2010, pp. 223-243.

[6] I. Verma, "W model of component based software development," *International Journal of advanced studies in Computer Science and Engineering*, vol. 3, no. 7, pp. 37-40, 2014.

[7] I. Crnkovic, M. Chaudron and S. Larsson, "Component-based development process and component lifecycle," in *International Conference on Software Engineering Advances*, Tahiti, 2006.

[8] B. Schätz, A. Pretschner, F. Huber and J. Philipps, "Model-based development of embedded systems," in *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, Montpellier, France, 2002.

[9] J. Eickhoff, A. Falke and H.-P. Röser, "Model-based design and verification—state of the art from Galileo constellation down to small university satellites," *Acta Astronautica*, vol. 61, no. 1-6, pp. 383-390, 2007.

[10] A. George, N. Psenjen, A. Gillette, J. Joji, T. Yavuz and C. Wilson, "Toward a suite of Middleware Services for Enhanced Spacecraft Configuration and Capability," in *Workshop on Spacecraft Flight Software*, Laurel, 2015.

[11] J. Samson, E. Grobelny, S. Driesse-Bunn, M. Clark and S. Van Portfliet, "New Millenium Program Space Technology 8 Dependable Multiprocessor: Technology and Technology Validation," *Journal of Spacecraft and Rockets*, vol. 49, no. 6, pp. 1043-1057, 2012.

[12] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku and W.-K. Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *International Conference on Intelligent Robots and Systems*, Edmonton, Canada, 2005.

[13] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku and W.-K. Yoon, "Composite component framework for RT-middleware (robot technology middleware)," in *International Conference on Advanced Intelligent Mechatronics*, Monterey, CA, 2005.

[14] T. Collett, B. MacDonald and B. Gerkey, "Player 2.0: toward a practical robot programming framework," in *Australasian Conference on Robotics and Automation*, Sydney, 2005.

[15] C. Schlegel, T. Hassler, A. Lotz and A. Steck, "Robotic software systems: from code-driven to model-driven designs," in *International Conference on Advanced Robotics*, Munich, 2009.

[16] A. Elkady and T. Sobh, "Robotics middleware: a comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, p. 15, 2012.

[17] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons and D. C. Schmidt, "Model driven middleware: a new paradigm for developing distributed real-time and embedded systems," *Science of Computer Programming, Special Issue on Foundations and Applications of Model Driven Architecture (MDA)*, vol. 73, no. 1, pp. 39-58, 2008.

[18] S. H. Pruett, G. J. Slutz, J. L. Paunicka and E. Portilla, "Hardware-In-The-Loop simulation using Open Control Platform," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Austin, Texas, 2003.

[19] L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J. Prasad, D. Schrage and G. Vachtsevanos, "An open platform for reconfigurable control," *Control Systems, IEEE*, vol. 21, no. 3, pp. 49-64, 2001.

[20] R. Ratcliff, S. T. LeDoux and W. W. Herling, "A modern CORBA-based approach to ad hoc distributed process orchestrations applied to MDO," in *Infotech@Aerospace*, Arlington, Virginia, 2005.

[21] U. Mittag, E. Schwartz, C. Lois and T. Rupp, "Towards a reusable micro control center for micro satellites," in *Space OPS 2004 Conference*, Montreal, 2004.

[22] A. Donati, G. Montroni, J. Eggleston and F. Zimmermann, "An ESA mission control system external interface to enable fast deployment of advanced operational functions," in *SpaceOps 2002 Conference*, Houston, 2002.

[23] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim and C.-H. Lee, "OPRoS: a new component-based robot software platform," *ETRI Journal*, vol. 32, no. 5, pp. 646-656, Oct 2010.

[24] A. Pasetti, W. Pree, J.-L. Terraillon and T. v. Overbeek, "An object-oriented component-based framework for on-board software," in *Proceedings of the Data Systems In Aerospace Conference*, Nice, 2001.

[25] M. Panunzio and T. Vardanega, "A component model for on-board software applications," in *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Lille, France, 2010.

[26] M. Panunzio and T. Vardanega, "A component-based process with separation of concerns for the development of embedded real-time software systems," *Journal of Systems and Software*, vol. 96, pp. 105-121, 2014.

[27] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka and J. Adamek, "A component-oriented framework for spacecraft on-board software," in *Proceedings of DASIA 2008, DAta Systems In Aerospace*, Palma de Mallorca, Spain, 2008.

[28] J. López, P. Royo, C. Barrado and E. Pastor, "Applying Marea middleware to UAS communications," in *Proceedings of the AIAA Infotech@Aerospace Conference and AIAA Unmanned Unlimited Conference 2009*, Seattle, 2009.

[29] A.-I. Rodríguez, F. Ferrero, E. Alaña, A. Jung, M. Panunzio, T. Vardanega and A. Grenham, "The component layer of COrDeT on-board software architecture," in *Proceedings of DASIA 2012*, Dubrovnik, 2012.

[30] B. Weps, "Entwicklung einer Schnittstelle für verteiles I/O in einer neuen on-board Computerarchitektur (in German)," Master Thesis, University of Applied Science Bingen, Bingen, Germany, 2013.

[31] S. Parkes and P. Armbruster, "SpaceWire: a spacecraft onboard network for real-time communications," in *14th IEEE-NPSS Real Time Conference*, Stockholm, 2005.

[32] O. Maibaum, D. Lüdtke and A. Gerndt, "Tasking framework: parallelization of computations in onboard control systems," in *Betriebssysteme für zukünftige Rechnerarchitekturen. Autumn Meeting: Special Interest Group Operating Systems, Gesellschaft für Informatik*, Berlin, Nov. 2013.

[33] B. S. Ainapure and S. S. Jadhav, Object oriented modeling and design, Pune, India: Technical Publications Pune, 2008.

[34] B. Councill and G. T. Heineman, "Definition of a software component and its elements," in *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001, p. 5–19.

[35] A. Kaur and K. S. Mann, "Component Based Software Engineering," *International Journal of Computer Applications*, vol. 2, no. 1, pp. 105-108, 2010.

[36] I. Crnkovic and M. Larsson, "Component-Based Software Engineering - new paradigm of software development," in *Proceedings of the Conference MIPRO 2001*, Opatija, 2001.

[37] A. Javed, "A model-driven framework for context-dependent component testing," Ph.D. Thesis, School of Information Technology and Electrical Engineering, University of Queensland, Queensland, 2014.

[38] R. R. Irish, J. L. Barker, S. N. Goward and T. Arvidson, "Characterization of the Landsat-7 ETM+ automated cloud-cover assessment (ACCA) algorithm," *Photogrammetric Engineering & Remote Sensing*, vol. 72, no. 10, pp. 1179-1188, 2006.

[39] V. Schaus, D. Lüdtke and A. Gerndt, "Advanced Spacecraft Systems Design using Model-based Techniques," in *2nd Federated Satellite Systems Workshop*, Moscow, Russian Federation, 2014.